18 | 磁盘视角：如果内存无限大，磁盘还有用武之地吗？

2021-04-23 吴磊

Spark性能调优实战 进入课程 >

讲述：吴磊
时长 16:26 大小 15.06M

你好，我是吴磊。

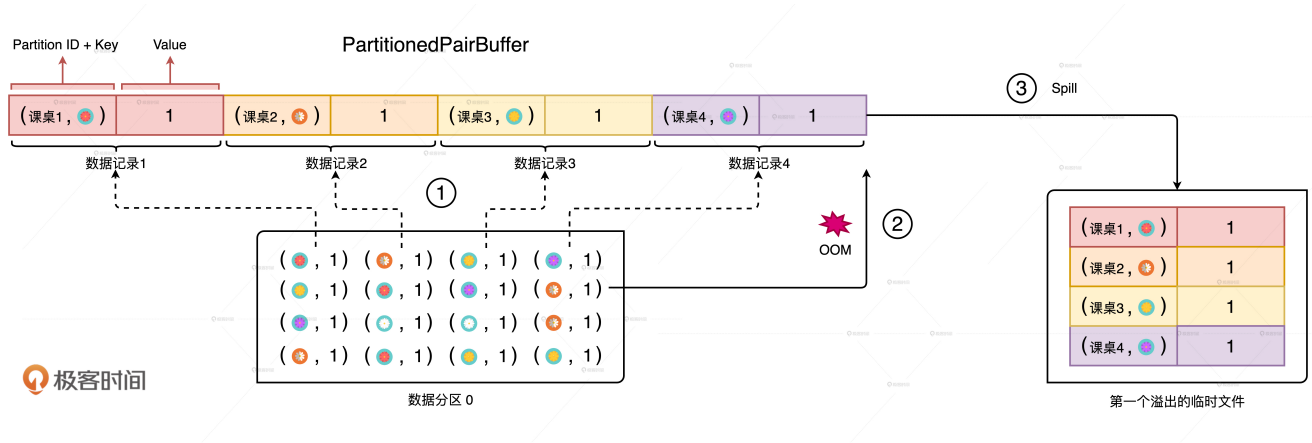
我们都知道，Spark 的优势在于内存计算。一提到“内存计算”，我们的第一反应都是：执行效率高！但如果听到“基于磁盘的计算”，就会觉得性能肯定好不到哪儿去。甚至有的人会想，如果 Spark 的内存无限大就好了，这样我们就可以把磁盘完全抛弃掉。当然，这个假设大概率不会成真，而且这种一刀切的思维也不正确。

如果内存无限大，我们确实可以通过一些手段，让 Spark 作业在执行的过程中免去所有的落盘动作。但是，无限大内存引入的大量 Full GC 停顿（Stop The World），很有可能会拖慢应用的执行性能，相比有磁盘操作的时候更差。这就不符合我们一再强调的，**调优的最终目的是在不同的硬件资源之间寻求平衡了**。

所以今天这一讲，我们就来说说磁盘在 Spark 任务执行的过程中都扮演哪些重要角色，它功能方面的作用，以及性能方面的价值。掌握它们可以帮助我们更合理地利用磁盘，以成本优势平衡不同硬件资源的计算负载。

磁盘在功能上的作用

在 Spark 当中，磁盘都用在哪些地方呢？在 Shuffle 那一讲我们说过，在 Map 阶段，Spark 根据计算是否需要聚合，分别采用 PartitionedPairBuffer 和 PartitionedAppendOnlyMap 两种不同的内存数据结构来缓存分片中的数据记录。分布式计算往往涉及海量数据，因此这些数据结构通常都没办法装满分区中的所有数据。在内存受限的情况下，溢出机制可以保证任务的顺利执行，不会因为内存空间不足就立即报 OOM 异常。



溢出数据到磁盘，避免频繁的OOM

以“仙女散花”的游戏为例，我们用 groupByKey 去收集不同花色的花朵。在 PartitionedPairBuffer 大小为 4 的情况下，当小红拿到的花朵数量超过 4 朵的时候，其余花朵要想进入内存，Spark 就必须把 PartitionedPairBuffer 中的内容暂时溢出到临时文件，把内存空间让出来才行。**这就是磁盘在功能上的第一个作用：溢出临时文件。**

当分区中的最后一批数据加载到 PartitionedPairBuffer 之后，它会和之前溢出到磁盘的临时文件一起做归并计算，最终得到 Shuffle 的数据文件和索引文件也会存储到磁盘上，也就是我们常说的 Shuffle 中间文件。**这就是磁盘的在功能上的第二个作用：存储 Shuffle 中间文件。**

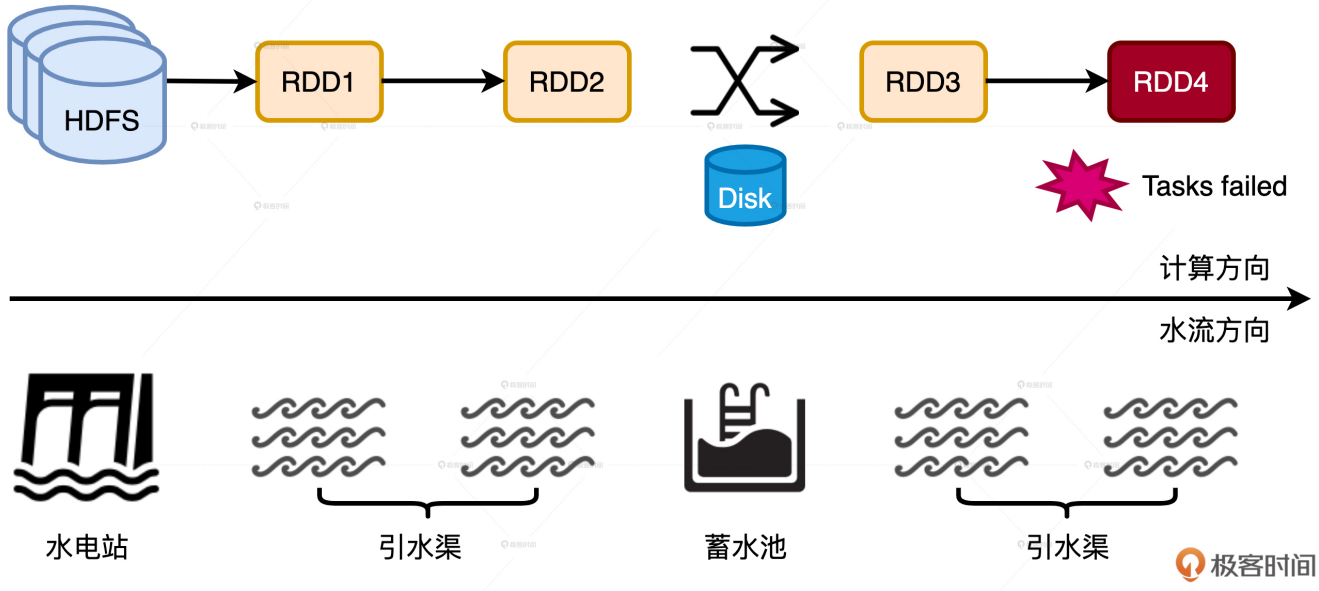
除此之外，**磁盘的第三个作用就是缓存分布式数据集。也就是说，凡是带*DISK*字样的存储模式，都会把内存中放不下的数据缓存到磁盘。**这些缓存数据还有刚刚讲的临时文件、中间文件，都会存储到 spark.local.dir 参数对应的文件系统目录中。

性能上的价值

在配置项那一讲我们说过，把 spark.local.dir 这个参数配置到 SDD 或者其他访问效率更高的存储系统中可以提供更好的 I/O 性能。除此之外，磁盘复用还能给执行性能带来更好的提升。所谓**磁盘复用，它指的是 Shuffle Write 阶段产生的中间文件被多次计算重复利用的过程**。下面，我就通过两个例子给你详细讲讲，磁盘复用的常见应用和它的收益。

失败重试中的磁盘复用

我们经常说，在没有 RDD Cache 的情况下，一旦某个计算环节出错，就会触发整条 DAG 从头至尾重新计算，这个过程又叫失败重试。严格来说，这种说法是不准确的。因为，失败重试的计算源头并不是整条 DAG 的“头”，而是与触发点距离最新的 Shuffle 的中间文件。



磁盘复用与蓄水池

我们以文稿示意图中的 DAG 为例子，HDFS 源数据经过两次转换之后，分别得到 RDD1 和 RDD2。RDD2 在 Shuffle 之后再进行两次计算，分成得到 RDD3 和 RDD4。

不幸的是，在计算 RDD4 的过程中有些任务失败了。在失败重试的时候，Spark 确实会从 RDD4 向前回溯，但是有了磁盘复用机制的存在，它并不会一直回溯到 HDFS 源数据，而是直接回溯到已经物化到节点的 RDD3 的“数据源”，也就是 RDD2 在 Shuffle Write 阶段输出到磁盘的中间文件。因此，**磁盘复用的收益之一就是缩短失败重试的路径，在保障作业稳定性的同时提升执行性能**。

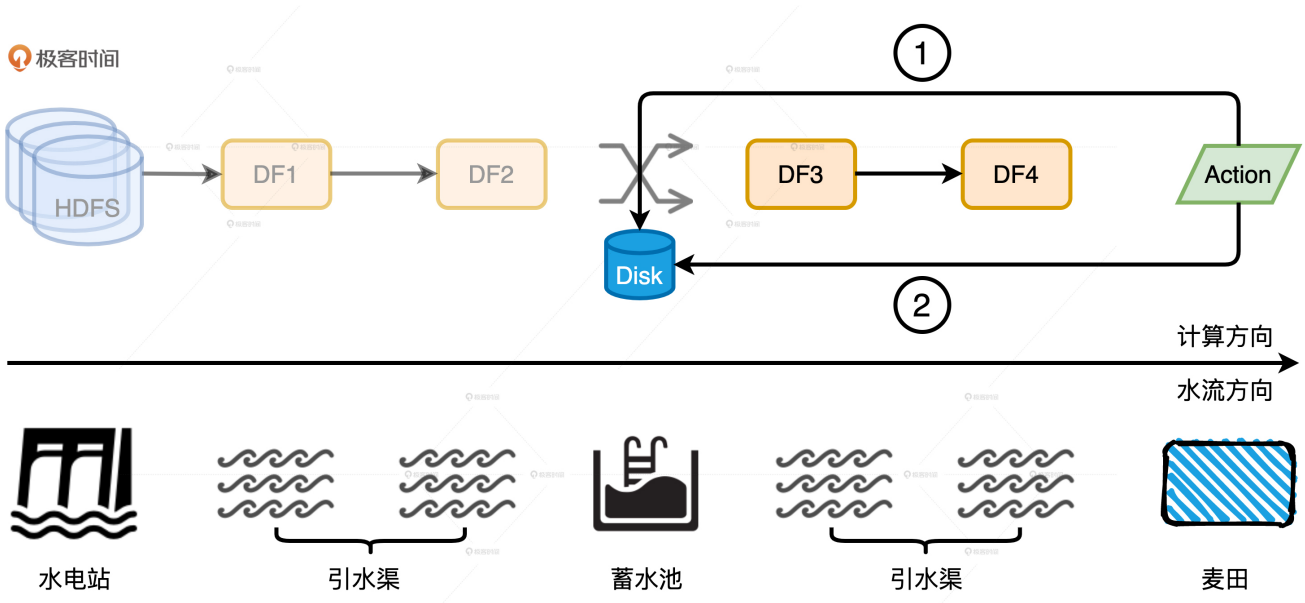
为了方便你理解，我们不妨把 DAG 中的流水线计算想象成是干渠灌溉，黄土高坡上的麦田一年到头也喝不到几滴雨水，完全依靠人工干渠进行灌溉。当水电站开闸放水的时候，水会沿着干渠一路向东流进支渠去滋养如饥似渴的麦苗。

一个水电站往往服务方圆百里大大小小的村子，如果每次灌溉都等着水电站开闸放水，遇上大旱的年头，水还没流到支渠，麦苗就都旱死了。要是能沿着干渠，每隔一段距离就修建一座蓄水池，那么附近的村民就能就近灌溉了。在这个干渠灌溉的类比中，水电站的水是 HDFS 数据源头，蓄水池就是 Shuffle 中间文件，就近取水、就近灌溉就是磁盘复用机制。

ReuseExchange 机制下的磁盘复用

你可能会说：“磁盘复用也没什么嘛，无非是在失败重试的时候，抄个近道、少走些弯路。在任务不出错的情况下是利用不到这项优势的。”没错，所以我们再来说说磁盘复用的另一种形式：ReuseExchange 机制。ReuseExchange 是 Spark SQL 众多优化策略中的一种，它指的是**相同或是相似的物理计划可以共享 Shuffle 计算的中间结果**，也就是我们常说的 Shuffle 中间文件。ReuseExchange 机制可以帮我们削减 I/O 开销，甚至节省 Shuffle，来大幅提升执行性能。

那我们该怎么有效利用 ReuseExchange 机制呢？在数据仓库场景下，为了得到数据报表或是可视化图表，用户往往需要执行多个相似的查询，甚至会把同样的查询语句执行多次。在这种情况下，ReuseExchange 策略在执行效率方面会带来非常大的收益。



同样或相似的查询利用ReuseExchange缩短执行路径

即便是在没有 DataFrame Cache 的情况下，相同或是相似的查询也可以利用 ReuseExchange 策略，在缩短执行路径的同时，消除额外的 Shuffle 计算。从数据复用的角度来说，ReuseExchange 和 DISK_ONLY 模式的 DataFrame Cache 能起到的作用完全等价。

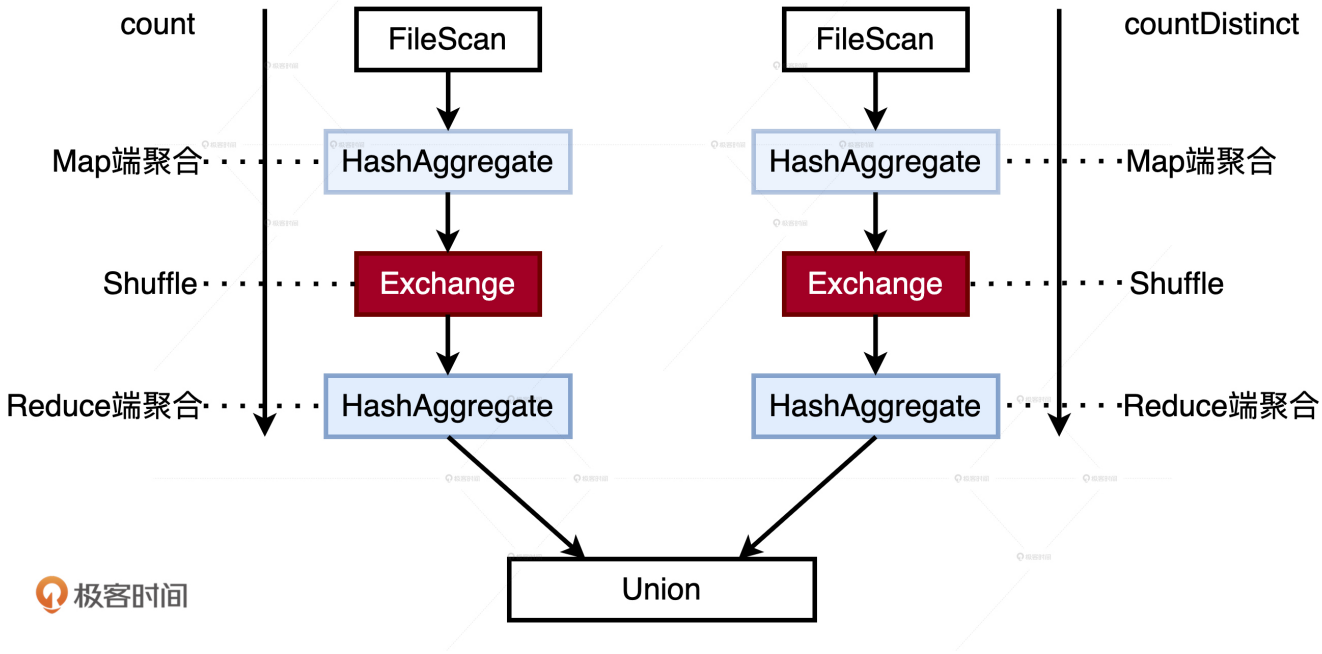
咱们来举个例子。现在有这样一个业务需求：给定用户访问日志，分别统计不同用户的 PV（Page Views，页面浏览量）、UV（Unique Views，网站独立访客），然后再把两项统计结果合并起来，以备后用。其中，用户日志包含用户 ID、访问时间、页面地址等主要字段。业务需求不仅明确也很简单，我们很快就能把代码写出来。

复制代码

```
//版本1：分别计算PV、UV，然后合并
// Data schema (userId: String, accessTime: Timestamp, page: String)

val filePath: String = _
val df: DataFrame = spark.read.parquet(filePath)

val dfPV: DataFrame = df.groupBy("userId").agg(count("page").alias("value")).w
val dfUV: DataFrame = df.groupBy("userId").agg(countDistinct("page").alias("va

val resultDF: DataFrame = dfPV.Union(dfUV)

// Result样例
| userId | metrics | value |
| user0  | PV      | 25 |
| user0  | UV      | 12 |
```

代码逻辑是先读取用户日志，然后在同一个 DataFrame 之上分别调用 count 和 countDistinct 计算 PV、UV，最后把 PU、UV 对应的两个 DataFrame 合并在一起。

虽然代码实现起来简单直接，但是，如果我们在 resultDF 之上调用 explain 或是通过 Spark UI 去查看物理计划就会发现，尽管 count 和 countDistinct 是基于同一份数据源计算的，但这两个操作的执行路径是完全独立的。它们各自扫描 Parquet 源文件，并且通过 Shuffle 完成计算，在 Shuffle 之前会先在 Map 端做本地聚合，Shuffle 之后会在 Reduce 端再进行全局聚合。



Parquet文件扫描两次、Shuffle两次

对于绝大多数的合并场景来说，计算流程大抵如此。显然，这样的做法是极其低效的，尤其是在需要合并多个数据集的时候，重复的数据扫描和分发就会引入更多的性能开销。那么，有没有什么办法，让同一份数据源的多个算子只读取一次 Parquet 文件，且只做一次 Shuffle 呢?

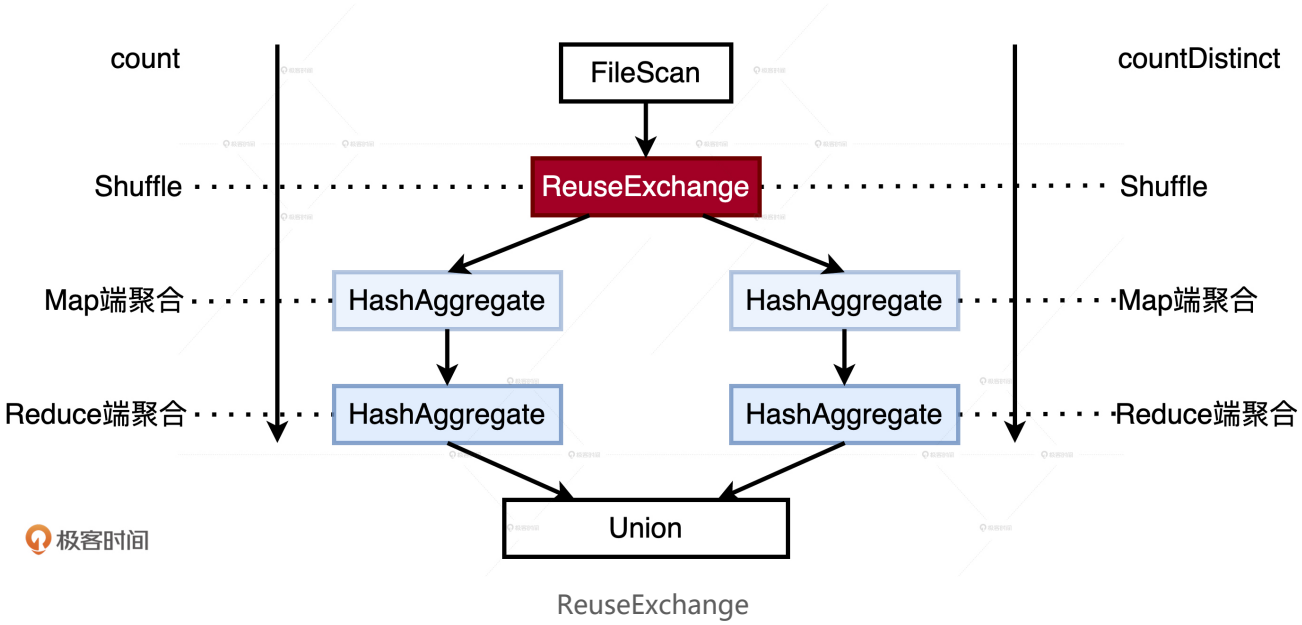
做了这么半天铺垫，答案自然是“有”。针对版本 1 中的代码，我们稍作调整就可以充分利用 ReuseExchange 策略来做优化。

复制代码

```
//版本2：分别计算PV、UV，然后合并
// Data schema (userId: String, accessTime: Timestamp, page: String)

val filePath: String = _
val df: DataFrame = spark.read.parquet(filePath).repartition($"userId")

val dfPV: DataFrame = df.groupBy("userId").agg(count("page").alias("value")).w
val dfUV: DataFrame = df.groupBy("userId").agg(countDistinct("page").alias("va

val resultDF: DataFrame = dfPV.Union(dfUV)

// Result样例
| userId | metrics | value |
| user0  | PV      | 25 |
| user0  | UV      | 12 |
```

需要调整的部分仅仅是数据源读取，其他部分的代码保持不变。在用 Parquet API 读取用户日志之后，我们追加一步重分区操作，也就是以 userId 为分区键调用 repartition 算子。

经过这个微小的改动之后，我们重新在 resultDF 之上调用 explain 或是查看 Spark UI 会发现，在新的物理计划中，count 或是 countDistinct 分支出现了 ReuseExchange 字样，也就是其中一方复用了另一方的 Exchange 结果。



ReuseExchange

通过观察执行计划不难发现，ReuseExchange 带来的收益相当可观，不仅是**数据源只需要扫描一遍，而且作为“性能瓶颈担当”的 Shuffle 也只发生了一次**。

另外，你可能也会发现，复用 Shuffle 中间结果的是两个不完全相同的查询，一个是用 count 做统计计数，另一个是用 countDistinct 做去重计数。你看，两个相似的查询，通过 ReuseExchange 数据复用，达到了使用 DISK_ONLY 缓存的等价效果。换句话说，你不需要手动调用 persist(DISK_ONLY)，也不需要忍受磁盘缓存的计算过程，就可以享受它带来的收益。这惊不惊喜、意不意外？

你可能会问：“既然 ReuseExchange 机制这么好用，满足什么条件才能触发 Spark SQL 去选择这个执行策略呢？”事实上，触发条件至少有 2 个：

多个查询所依赖的分区规则要与 Shuffle 中间数据的分区规则保持一致

多个查询所涉及的字段（Attributes）要保持一致

对于第一个条件，我们在案例中已经演示过了，两个查询都用 userId 分组，这就要求所依赖的数据必须要按照 userId 做分区。这也是为什么我们在版本 2 的代码中，会添加以 userId 为分区键的 repartition 算子，只有这样，Shuffle 中间结果的分区规则才能和查询所需的分区规则保持一致。

仔细观察 count 和 countDistinct 两个查询所涉及的字段，我们会发现它们完全一致。实际上，如果我们把 count 语句中的`count("page")`改为`count("*")`也并不影响 PV 的计算，但是，看似无关痛痒的改动会导致第二个条件不能满足，从而无法利用 ReuseExchange 机制来提升执行性能。版本 2 中的`count("page")`改为`count("*")`之后，物理计划会回退到版本 1，我把其中的变化留给你作为课后作业去对比。

小结

磁盘虽然在处理延迟上远不如内存，但在性能调优中依然不可或缺。理解磁盘在功能上和性能上的价值，可以帮助我们更合理地利用磁盘，以成本优势平衡不同硬件资源的计算负载。

从功能上看，磁盘在 Spark 中主要有 3 方面的作用，分别是溢出临时文件、缓存分布式数据集和存储 Shuffle 中间文件。这 3 方面功能在提升作业稳定性的同时，也为执行效率的提升打下了基础。

从性能上看，利用好磁盘复用机制，可以极大地提高应用的执行性能。磁盘复用指的是 Shuffle Write 阶段产生的中间文件被多次计算重复利用的过程。磁盘复用有两种用途，一个是失败重试，另一个是 ReuseExchange 机制。其中，失败重试指的就是任务失败之后尝试重头计算。这个过程中，磁盘复用缩短了失败重试的路径，在保障作业稳定性的同时，提升执行性能。

ReuseExchange 策略指的是，相同或是相似的物理计划可以共享 Shuffle 计算的中间结果。ReuseExchange 对于执行性能的贡献相当可观，它可以让基于同一份数据源的多个算子只读取一次 Parquet 文件，并且只做一次 Shuffle，来大幅削减磁盘与网络开销。

不过，要想让 Spark SQL 在优化阶段选择 ReuseExchange，业务应用必须要满足 2 个条件：

多个查询所依赖的分区规则要与 Shuffle 中间数据的分区规则保持一致

多个查询所涉及的字段要保持一致

每日一练

1. 请你把 count 计算中的`count("page")`改为`count("*")`，以此来观察物理计划的变化，并在留言区说出你的观察
2. 为了触发 ReuseExchange 机制生效，我们按照 userId 对数据集做重分区，结合这一点，你不妨想一想，在哪些情况下，不适合采用 ReuseExchange 机制？为什么？

期待在留言区看到你的思考和答案，我们下一讲见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 17 | 内存视角（三）：OOM都是谁的锅？怎么破？

下一篇 19 | 网络视角：如何有效降低网络开销？

精选留言 (9)

写留言

sky_sql
2021-04-23

老师好！咨询个题外话，在调度系统中讲到SchedulerBackend 用 ExecutorData 对 Executor 进行资源画像，ExecutorData中有RPC 地址、主机地址、可用 CPU 核数和满配 CPU 核数等等，但是没有内存相关的，是spark的Executor 内存这块比较复杂?
对比hadoop在调度时会考虑内存是否满足吗?

展开

作者回复: 好问题~ 我们还是分资源调度和任务调度两种情况来说。

Spark在做任务调度之前，SchedulerBackend封装的调度器，比如Yarn、Mesos、Standalone，实际上已经完成了资源调度，换句话说，整个集群有多少个containers/executors，已经是一件确定的事情了。而且，每个Executors的CPU和内存，也都是确定的了（因为你启动Spark集群的时候，使用配置项指定了每个Executors的CPU和内存分别是多少）。资源调度器在做资源调度的时候，确实是同时需要CPU和内存信息的。

资源调度完成后，Spark开始任务调度，你的问题，其实是任务调度范畴的问题。也就是TaskScheduler在准备调度任务的时候，要事先知道都有哪些Executors可用，注意，是可用。也就是TaskScheduler的核心目的，在于获取“可用”的Executors。

现在来回答你的问题，也就是：为什么ExecutorData不存储于内存相关的信息。答案是：不需要。一来，TaskScheduler要达到目的，它只需知道Executors是否有空闲CPU、有几个空闲CPU就可以了，有这些信息就足以让他决定是否把tasks调度到目标Executors上去。二来，每个Executors的内存总大小，在Spark集群启动的时候就确定了，因此，ExecutorData自然是没必要记录像Total Memory这样的冗余信息。

再来说Free Memory，首先，我们说过，Spark对于内存的预估不准，再者，每个Executors的可用内存都会随着GC的执行而动态变化，因此，ExecutorData记录的Free Memory，永远都是过时的信息，TaskScheduler拿到这样的信息，也没啥用。一者是不准，二来确实没用，因为TaskScheduler拿不到数据分片大小这样的信息，TaskScheduler在Driver端，而数据分片是在目标Executors，所以TaskScheduler拿到Free Memory也没啥用，因为它也不能判断说：task要处理的数据分片，是不是超过了目标Executors的可用内存。

综上，ExecutorData的数据结构中，只保存了CPU信息，而没有记录内存消耗等信息。不知道这些能不能解答你的问题？有问题再聊哈~

4

zxk
2021-04-23

老师，如果我在 df.groupBy("userId") 后做磁盘 cache ，后续再做两个 agg 操作，是否跟 ReuseExchange 效果一样？如果一样的话，那么这时候是不是可以不用管 agg 里涉及的字段了？

展开

作者回复: 对，如果用磁盘Cache的话，只要把“最小公共子集”都Cache住就行了~ ReuseExchange虽然效果相同，但是前提条件确实比较苛刻，不容易命中。所以稳妥的话，还是使用Disk Cache。

1 3

kingcall
2021-04-27

感觉我们只要能让算子复用shuffle 后的数据就可以做到减少shuffle,但是文中提到ReuseExchange条件太苛刻，而评论中说的缓存groupBy后的结果也就是shuffle 后的结果，供后续多个聚合统计又做不到，哈哈，这是不是spark 后续可以优化的地方
我觉得想想为啥spark 没有给RelationalGroupedDataset 添加cache操作，理论上来说shuffle 后的数据和shuffle 前的数据量是一致的，只要shuffle前的数据可以缓存下来，sh...

展开

作者回复: 第一个问题，可以参考我留给@ Geek_d794f8 同学的答复哈，应该就在你这条留言的下面~

关于第二个问题，我理解你的问题是，为什么groupBy之后，不能加cache，或者说，没有cache API。其实你已经给出答案啦~ 也就是你说的1、2。我同意你的说法，分组这种操作，仅仅是聚合的前奏，换句话说，如果没有聚合，很多场景是根本不需要分组的。所以把使用频率极低的数据做缓存，没有意义。

1

蠟筆小嘶
2021-04-25

内存无限大的情况下，对象不会遇到放不下的问题，为什么会有GC呢。。。。

展开

作者回复: 内存无限大，JVM的配置参数xmx、xms是有界的呀 😀

1 1

RespectM
2021-04-25

老师好，可以讲讲企业级Spark的机器硬件该如何选行吗，以及应该考虑哪些问题。网上都是说根据需求，有什么根据吗，能不能举个例子。

展开

作者回复: 确实是看需求、看场景哈，没有一个统一的标准，不过，还是有一些General的Guidelines可以遵循，核心思路还是“消除瓶颈”。硬件的选型，取决于你的计算负载类型，也就是计算密集型？内存密集型？还是磁盘、网络密集型？

我们来分类讨论：
1. 如果你的计算场景涉及到大量的聚合、排序、哈希计算、数值计算，等等，那么你的机器配置就要加强CPU
2. 如果你的计算场景需要反复消耗同一份或是同一批数据集，比如机器学习、数据分析、图计算，那么，为了把需要频繁访问的数据缓存进内存，你自然需要加大内存配置。
3. 如果你的计算场景会引入大量shuffle，又不能通过广播来消除Shuffle，那么你就需要配置足够的SSD以及高吞吐网络。

解答得比较粗糙，并没有具体可操作的建议，都是一些Guidelines，希望能帮到你~

1 1

Fendora范东_
2021-04-23

有个疑问
有reuseexchange算子的执行计划中，既然reuseexchange阶段是shuffle，shuffle后面直接就应该是reduce聚合了，为啥在这两者之间还存在map端聚合？

展开

作者回复: ReuseExchange指的是，后面相似的计算逻辑复用Shuffle计算的中间文件，也就是Shuffle map阶段输出到本地磁盘的中间文件，这和Shuffle的Map端聚合并不冲突，实际上，直白地说，是没什么关系

aof
2021-05-05

第二题，想到有一种情况：假如原始日志数据中，某些用户的访问日志比较多，而且在调用repartition()哈希分区之后，恰好都落到了一个分区，那后续的计算可能会出现数据倾斜。其他的没想到哈哈

展开

Geek_d794f8
2021-04-26

老师，df.groupBy("userId").agg(count("page")和df.groupBy("userId").agg(countDistinct("page")都会按照userId的hashcode值进行Hash Partitioner。而repartition($"userId")，只不过是在这之前进行了Hash Partitioner的shuffle操作。本质上是一样的呀？为什么最后一个会reuseExchange，一个不会呢？

展开

作者回复: 好问题，这里确实有点Tricky。

这部分其实需要一些“前置引用”的知识，这些知识其实是在22讲Physical Planning才会涉及，所以在这一讲这里会比较难理解。

是这样的，Spark SQL执行计划中的每一个节点，都有4个重要的属性，分别是：
节点要求的输入：
1）requiredChildDistribution
2）requiredChildOrdering
节点的输出：
3）outputPartitioning
4）outputOrdering

每个节点都有输出的分区情况、排序情况，也就是3）、4）；同时，每个节点对于自己的子节点，都有关于 分区和排序的要求，也就是1）、2）。

当子节点的分区与排序情况，不满足当前节点的输入要求时，Spark SQL就会在Physical planning阶段，强行插入一些中间节点，比如Exchange（Shuffle）。

回答你的问题：咱们的例子中，两个groupBy，都是从Parquet读源数据，这个时候，读取Parquet的Scan节点，它的outputPartitioning，和groupBy这个节点对于输入的要求，两者是不matc

h的，因此，Spark SQL会强行插入Exchange操作符。

尽管我们肉眼能看出来，两个groupBy需要的shuffle是一样的，但是，Spark SQL可不知道，这件事对它来说，是透明的。它没有任何渠道，去知道说：“OK，他俩是一样的，我可以复用”

repartition的意义在于，它一下变成了两个groupBy共同的依赖，repartition之后的DataFrame，它的outputPartitioning，刚好都是groupBy节点所需要的，因此Spark SQL回溯到这里的时候，就会发现：“我去，之前的计算已经满足我的requiredChildDistribution，我直接拿来用就好了。”

逻辑就是这么个逻辑，可能一两句话没说清楚，可以重点关注22讲的EnsureRequirements那部分，那部分会咱们这里交代的逻辑做详细展开哈~

1

斯盖丸
2021-04-23

ReuseExchange的第二个条件也太苛刻了吧，难道多选级列就不行了，实用性一下子下来了...

展开

作者回复: 确实比较苛刻~